ELEC-4200

Digital System Design

FROM: Walker McGilvary

TO: Professor of ELEC 4200

DATE: December 22, 2024

LAB SECTION: Section 001

Final Project - Basic CPU

Introduction

This project implements a basic CPU (central processing unit) design using Verilog. This CPU has three main modules: **CPU, ALU,** and **Seven-Segment-Decoder**. The CPU module is the top-level module which instantiates the ALU and Decoder. Additionally, the CPU cycles through three different states: **FETCH**, **EXECUTE**, and **UPDATE**.
The ALU has two inputs `alu_in1` and `alu_in2`, and outputs `alu_out`.
The seven-segment-decoder is responsible for outputting the proper seven-segment display LEDs to represent a BCD number.

In addition to the Verilog CPU, I implemented a custom assembly language, which is converted to 16-bit machine code using a Python script. This greatly simplifies the programming process.

CPU Architecture

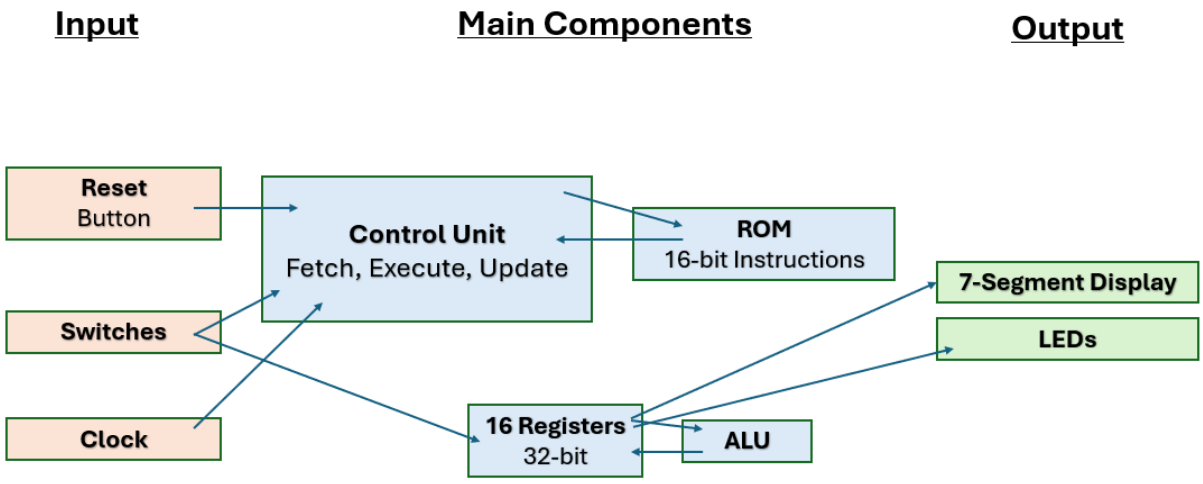

Figure 1: Block Diagram of CPU

The CPU consists of four main components to execute instructions. The Control Unit has three stages to run each line of code. The Program ROM stores up to 256 16-bit instructions that are fetched and decoded by the Control Unit. The registers contains sixteen 32-bit general-purpose registers, which store data for computation and results. The ALU (Arithmetic Logic Unit) performs operations like addition, subtraction, and equality comparison.

Input is handled through the FPGA's switches, which can load values directly into registers. Output is managed through a 7-segment display controller that shows register values. The entire system is synchronized by the clock signal and can be reset to its initial state.

This architecture, while simple, provides all necessary components for a functional CPU capable of

executing basic programs and algorithms.

Key Features

Key features include:

- 32-bit registers
- 16 general-purpose registers
- Three-stage execution cycle:
  - Fetch: Load current instruction from program memory
  - Execute: Perform instruction operation (ALU, register transfer, etc.)
  - Update: Increment program counter and update outputs
- Opcodes:
  - MOV: Register-to-register transfer
  - LOAD: Load immediate 12-bit value
  - ALU operations: Addition, subtraction, equality comparison
  - Branch operations for program flow control
  - I/O instructions for FPGA board interaction
- Custom assembly language with Python-based assembler
- FPGA Integration:
  - 7-segment display output
  - Switch inputs
  - LED outputs for debugging
- 256 lines of instructions, each 16-bit

Limitations

Because this CPU design is very simple, there are some limitations:

- Memory Constraints:
  - Maximum program size of 256 instructions
  - Only 16 general-purpose registers
- Instruction Limitations:
  - LOAD instruction can only handle 12-bit immediate values, despite having 32-bit registers

  - Basic branching (only `branch_if_zero` implemented)

- Performance:
  - Three clock cycles required per instruction
  - No pipelining or optimization features

However, despite these limitations, this CPU is still very powerful.

Assembly Language and Parser

The CPU uses a custom assembly language, which greatly simplifies the programming of the CPU. The instruction set consists of seven opcodes. The first three bits (15-13) of each instruction specify the opcode, with the remaining bits used for opcode-specific instructions.

Instruction Format and Opcodes

- MOV (000): Copy value between registers
  - Example: `mov 1 2` // Copy from register 1 to register 2
- LOAD (001): Load 12-bit immediate value into register
  - Example: `load 0 5` // Load value 5 into register 0
- ALU (010): Perform arithmetic operation
  - Example: `alu 0 1 add` // Add registers 0 and 1
- SAVE_ALU (011): Store ALU result in register
  - Example: `save_alu 2` // Save ALU result to register 2
- LOAD_INPUT (100): Read from FPGA switches
- BRANCH_IF_ZERO (101): Conditional branching
- END (111): Terminate program execution

Python Parser

A Python script handles the conversion from assembly language to machine code. The parser:

- Reads assembly code line by line
- Ignores comments (lines starting with //)
- Converts each instruction into its 16-bit binary representation
- Handles different instruction formats based on opcode
- Outputs machine code to a program.mem file

For example, the Fibonacci sequence can be programmed with this assembly language:

```
// FIBONACCI SEQUENCE
load 0 1          // first variable
load 1 1          // second variable
load 2 0          // constant = 0

mov 0 15          // OUTPUT CURRENT NUMBER (register 15 goes to basic_output)

alu 0 1 add       // add first & second variables
save_alu 4        // save output to the fourth register

mov 1 0           // set first variable = second variable
mov 4 1           // set second variable = current fibbonaci number

branch_if_zero 2 2       // loop indefinitely (reg 0 is a constant equal to 0)

load 14 1                // it should never here, but if it does its bugged
end
```

The programming process is seen in the figure below:

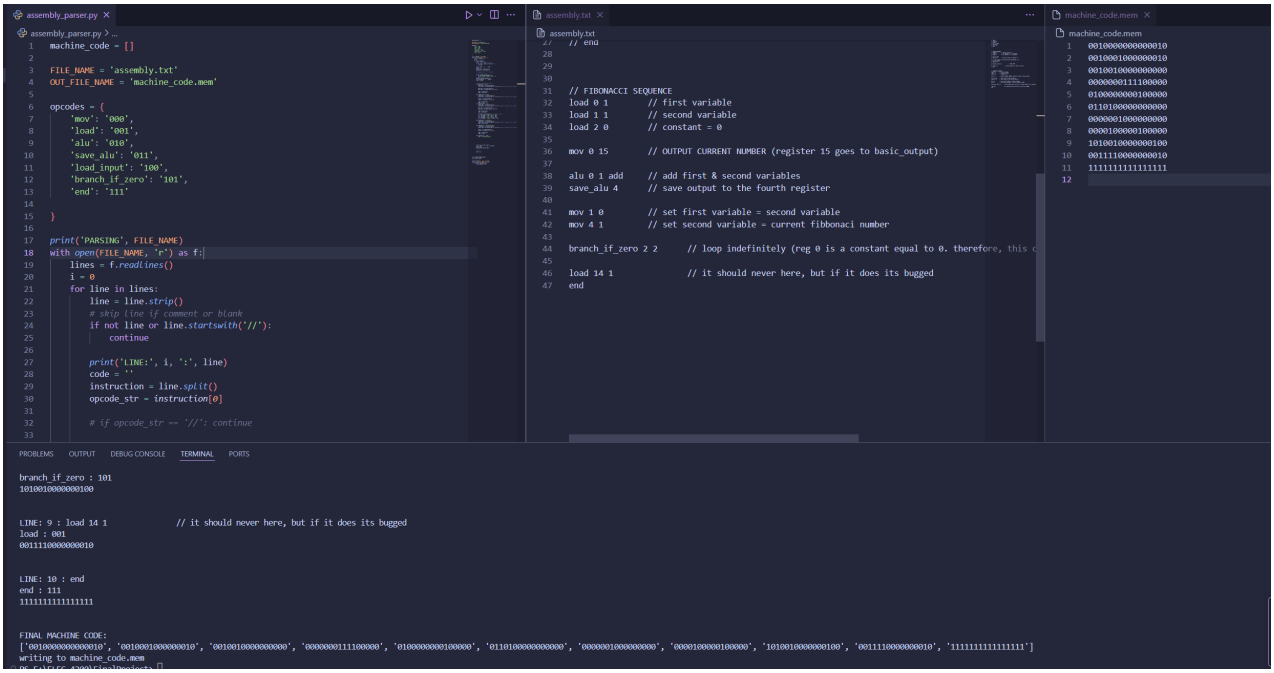

Figure 2: Fibonacci Sequence - Python Parser, Assembly Code, and Machine Code

Simulation

Using a testbench, the CPU operation can be tested in Vivado.

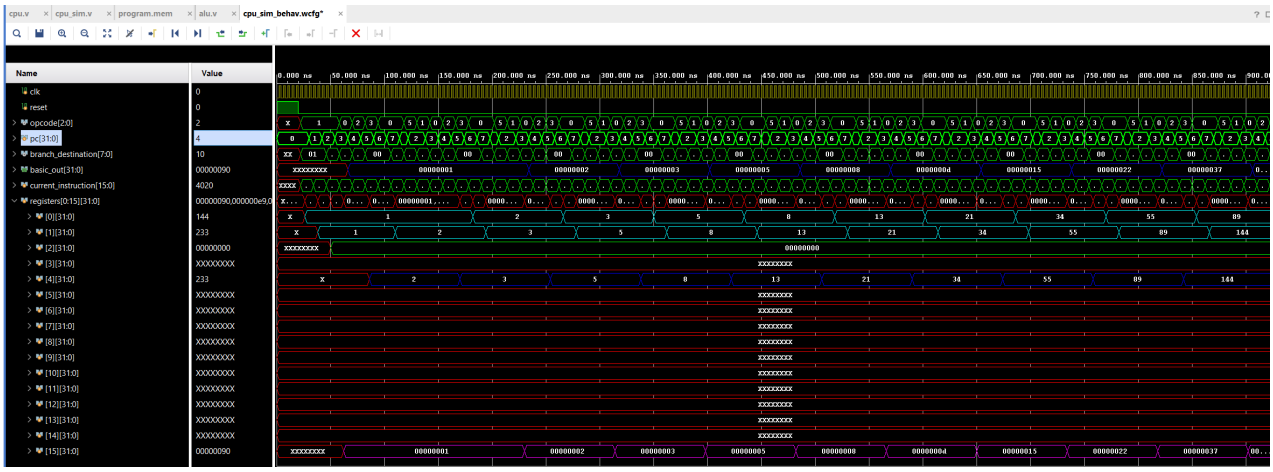

Figure 3: Fibonacci Sequence - Vivado Simulation

The figure above shows the Vivado simulation of the Fibonacci Sequence program.
Key observations:

- 3 clock cycles per line of code (corresponds to 3 CPU states)
- Registers 0 and 1 are added, and register 4 acts as an accumulator of the sequence.
- Register 15 is output to the 7-segment display.
- The program keeps looping, which is due to the `branch_if_zero` statement.

The figure below shows the simulation of a countdown program:

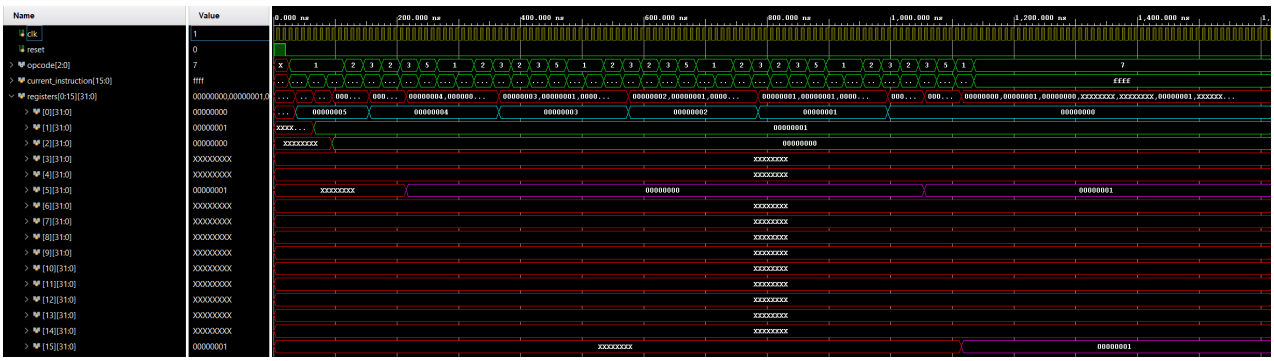

Figure 4: Countdown - Vivado Simulation

- Register 0 starts at 5
- Register 0 decrements to 0.

- When the register reaches 0, the `end` opcode is read, and the program is finished.

This program was simply programmed with the `alu 0 1 sub` operation (subtract 1 from register 0).

Challenges of FPGA Integration

Next, with the logical verification of the CPU operation (testbench simulations), the CPU will be tested on the FPGA hardware. Because of how Vivado synthesizes and implements the Verilog code, this process took a lot of debugging to achieve correct operation on the FPGA board.

The largest challenge was the 7-segment display clock. The 7-segment display shows multi-digit numbers by quickly rotating between digits. Each digit is briefly lit up in sequence, cycling fast enough that the human eye perceives all digits as continuously lit. This required a separate clock divider to control the display's switching speed.
However, because of the separate clock and clock divider, Vivado didn't synthesize this as expected. When the 7-segment display clock was finally working, I was happy to find that the CPU synthesized perfectly, and could run counter programs, the Fibonacci program, and an addition program (using switches as inputs).

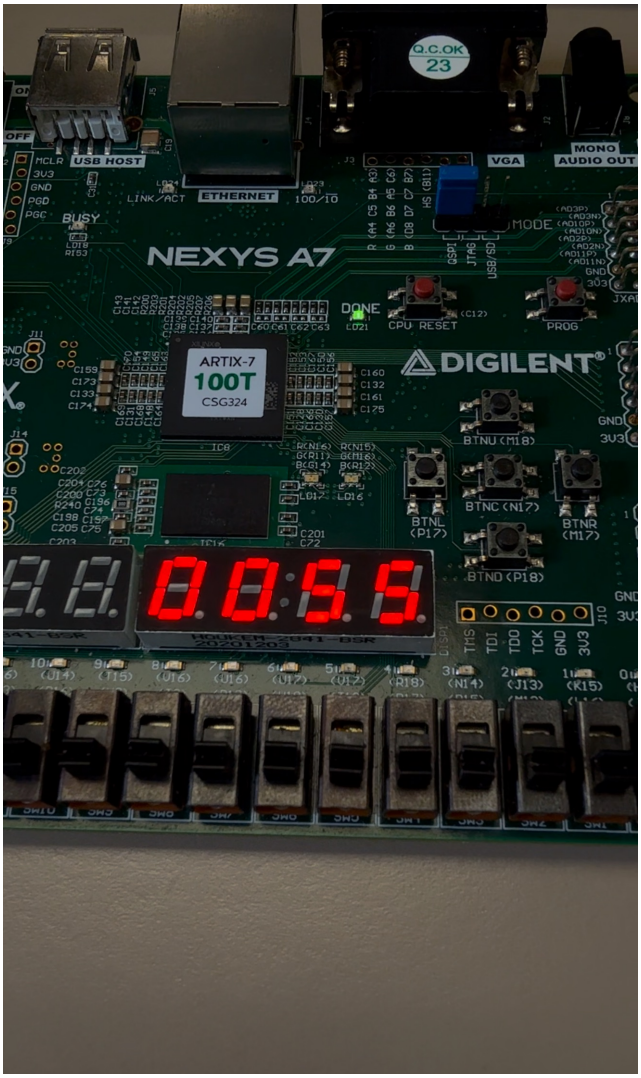

Figure 5: 11th Digit in Fibonacci Sequence

As seen above, the Fibonacci Sequence program worked as intended on the hardware. After many clock cycles, the digit `55` is displayed on the 7-segment display.

Conclusion

This concludes my Final Design Project for ELEC 4200 - Digital System Design. I implemented a basic CPU, written in Verilog. While simple, this CPU is very powerful, and has enough opcodes to run almost any algorithm. The multi-cycle design, with its Fetch, Execute, and Update stages, allow for a fully functional CPU. The instruction set, though minimal, supports essential operations like data movement, arithmetic, branching, and I/O interaction.

The project combines Verilog HDL, custom assembly language development, and practical FPGA implementation. Working example programs, including the Fibonacci sequence and switch-input calculator,

prove the CPU's computational capabilities. Despite limitations like 12-bit immediate values and 256-instruction memory, the design is Turing complete and should be able to run almost any algorithm.

The project code is on GitHub: https://github.com/TheSlabby/minimal-cpu